

SDEV1201

Database Fundamentals

LESSON 17

Let's review

Anonymous Code Blocks

Why use DO?

Run multi-statement logic without creating a stored function/procedure.

Perfect for data fixes, quick loops, conditional DDL, and ad-hoc scripting.

Run multi-statement logic without creating a stored function/procedure.

Anonymous Code Blocks

--\$\$ create the block for the code that you will execute

DO \$\$

Declare --A section to declare variables

my_message TEXT:= 'Hello World!'; --Declares a variable called my_message and assigns 'Hello World to it'

my_mood TEXT:= 'groovy';

Begin--The executable statements go here

Raise Notice '% Hope you have a % day!', my_message,my_mood;

--Raises a message to pgamdins messages tab or whatever output console you are working in. It is mainly for testing and debugging.

--% is a place holder

--You can also format a raise Notice as

Raise Notice using message = my_message || ' Hope you have a ' || my_mood || 'day!';

End \$\$;

Variables

Variables

Variables are used to hold and manipulate values in memory. They must be declared before they are used, allowing PostgreSQL to allocate storage efficiently.

Syntax

Declare

```
variablename datatype;  
variablename datatype;  
variablename datatype;
```

All variables must be declared at the beginning of the block, before any executable statements.

Variables

Assigning Values to Variables

Variables can be assigned literal values or values from queries.

Assigning a Literal Value

```
variable_name := literal_value;
```

Variables

Assigning a Value from a Query

```
SELECT column_name  
INTO    variable_name  
FROM    table_name  
WHERE   condition;
```

Flow Control

You can control the execution of SQL statements using flow control structures such as If, Elself (optional), and Else (optional). You can have as many Elself statements as you want, but only one Else statement is allowed. You can use Begin and End for readability, but it is not required in If statements.

Flow Control

Syntax:

If **condition** Then

Begin

-- SQL statements

End

Elsif **another_condition** Then

Begin

-- SQL statements

End

Else

Begin

-- SQL statements

End

End If;

Flow Control

Anonymous Code Blocks (Do)

Each Do block or stored procedure executes as a single unit.

You can separate logical sections of a script simply by executing one block at a time.

Hypothetical Example (Client table does not exist in IQSchool):

Do \$\$

Begin

 Alter Table Client

 Add Column Active Char(1);

End \$\$;

Do \$\$

Begin

 Alter Table Client

 Add Constraint CK_Client_Active_TF Check (Active In ('T', 'F'));

End \$\$;

In this example, each Do block executes separately. This ensures the column is added before the constraint is created.

Stored Procedures

A **stored procedure** is a **set of SQL statements**. It is very similar to other procedures that you may have used in other languages. The stored procedure has a name and can accept parameters.

A stored procedure in PostgreSQL is a named block of SQL statements that can be executed as a single unit. Stored procedures allow you to group related logic together, accept parameters, and control program flow; much like procedures in other programming languages.

They're compiled into machine language and stored in the database.

After it's been created, a SP can be executed without having to be re-compiled each time as a script would.

Stored Procedures

Simplified Syntax

In PostgreSQL, stored procedures are created using the CREATE PROCEDURE statement. They are written using the PL/pgSQL procedural language.

Syntax:

```
CREATE PROCEDURE procedure_name ( [parameter_list] )  
LANGUAGE plpgsql  
AS $$body$$  
BEGIN  
    -- SQL statements  
END;  
$$body$$;
```

Stored Procedures

Stored Procedure Capabilities

A PostgreSQL stored procedure can include:

- **DML statements** (INSERT, UPDATE, DELETE, SELECT)
- **Flow control** (IF, ELSIF, LOOP, etc.)
- **Transactions** (procedures can explicitly COMMIT or ROLLBACK)
- **Exception handling** (BEGIN ... EXCEPTION ... END)
- **Variable declarations and assignments**

Stored Procedures

Modifying a Stored Procedure

You can update an existing procedure in one of two ways:

1. Drop and Recreate

Dropping removes the procedure completely from the database.

```
DROP PROCEDURE procedure_name ( [parameter_types] );
```

Then recreate it using CREATE PROCEDURE.

Stored Procedures

2. Use **ALTER PROCEDURE**

You can modify some attributes (such as the owner or schema) or replace the procedure definition using **CREATE OR REPLACE PROCEDURE**.

```
CREATE OR REPLACE PROCEDURE procedure_name ( [parameter_list] )  
LANGUAGE plpgsql  
AS $body$  
BEGIN  
    -- Updated SQL statements  
END;  
$body$;
```


Example

Invoking Stored Procedures

To execute a procedure:

```
CALL procedure_name ( [arguments] );
```

Example:

```
Call PR_UpdateWithdrawStatus();
```

Functions

If you need to return a value, you will want to use a function instead. Stored functions are created using the CREATE FUNCTION statement. They are also written using the PL/pgSQL procedural language.

Functions

Simplified Syntax:

```
CREATE FUNCTION function_name ( [parameter_list] )
```

```
Returns data_type
```

```
LANGUAGE plpgsql
```

```
AS $$
```

```
BEGIN
```

```
    -- SQL statements
```

```
    Return value;
```

```
END;
```

```
$$;
```

Functions

Invoking Stored Function

To run a function:

```
Select FN_GetTotalPayments();
```

Today's Objective

By the end of this section you will be able to:

- ❑ Write **stored procedures and functions with parameters**
- ❑ Stored Procedure/Function **Error Handling**

Documentation can be found in Brightspace in the “Week 10 & 11 – Stored Procedures” section under Materials. “Stored Procedures and Functions – Parameters (PostgreSQL Version)” and “Stored Procedure and Function Error Handling (PostgreSQL)”.

Parameters

Parameters in a PostgreSQL stored procedure (or function) allow values to be passed in and used inside the procedure (or function). These parameters can be used for filtering data, performing calculations, or controlling logic within SQL statements.

Parameters

In PostgreSQL, parameters are declared inside parentheses after the procedure name.

Each parameter must have:

- A name (which starts with a colon or nothing at all — not @ as in SQL Server),
- A data type, and
- An optional default value can be assigned (for example, Default Null or = Null), which is used if no value is provided when calling the procedure or function.
- To test whether a parameter was passed, use Is Null, not = Null.

Example: Procedure with a Parameter

The following example creates a stored procedure that accepts a Student_ID parameter:

```
Create Or Replace Procedure PR_LookupStudent
(p_StudentID integer = Null)
Language plpgsql
As $body$
Begin
    If p_StudentID Is Null Then
        Raise Notice 'StudentID was not passed in';
    Else
        Raise Notice 'Looking up student with Id: %', p_StudentID;
    End If;
End;
$body$;
```


Executing a Stored Procedure with a Parameter

You can execute a stored procedure using the CALL statement:

```
Call PR_LookupStudent(12345);
```

Or, if you want to use the default parameter value:

```
Call PR_LookupStudent ();
```

Example: Procedure with multiple Parameters

Parameters are passed in order, separated by commas.

```
Create Procedure PR_AddStudent (p_StudentID integer, p_FirstName varchar, p_LastName varchar,  
p_Gender char, p_BirthDate date)  
Language plpgsql  
As $body$  
Begin  
    Insert Into student (StudentID, FirstName, LastName, Gender, BirthDate)  
        Values (p_StudentID, p_FirstName, p_LastName, p_Gender, p_BirthDate);  
End;  
$body$;
```

Call the procedure with multiple parameters

```
Call PR_AddStudent (100, 'Jane', 'Doe', 'F', '01-Jan-2000');
```

Example: Function with a Parameter

Here's a stored function that retrieves a student's full name based on a passed in StudentID. If no value is passed in, it returns a message instead of the student's name.

```
Create Function FN_Get_Student_Name (p_StudentID integer = Null)
```

```
Returns varchar
```

```
Language plpgsql
```

```
As $body$
```

```
Declare v_StudentName varchar(80);
```

```
Begin
```

```
    If p_StudentID Is Null Then
```

```
        Return 'StudentID was not passed in';
```

```
    Else
```

```
        Select FirstName || ' ' || LastName
```

```
        Into v_StudentName
```

```
        From Student
```

```
        Where StudentID = p_StudentID;
```

```
        Return v_StudentName;
```

```
    End If;
```

```
End;
```

```
$body$;
```

Run the function with a Parameter (or not)

Select FN_Get_Student_Name(199899200);

-- or --

Select FN_Get_Student_Name();

Activity

Create a procedure named PR_RegisterCourse which registers a student into a course. The parameters are StudentID, CourseID, Semester, and StaffID.

```
Create Procedure PR_RegisterCourse (p_StudentID integer, p_CourseID char(8), p_Semester char(5), p_StaffID  
smallint)
```

```
Language plpgsql
```

```
As $body$
```

```
Begin
```

```
    Insert Into Registration (StudentID, CourseID, Semester, StaffID)
```

```
        Values (p_StudentID, p_CourseID, p_Semester, p_StaffID);
```

```
End;
```

```
$body$;
```

```
Call PR_RegisterCourse (199966250, 'COMP1017', '2000S', 6);
```

```
Call PR_RegisterCourse (199966250::integer, 'COMP1017'::char(8), '2000S'::char(5), 6::smallint);
```

Activity

Retrieve the highest payment amount for a given year.

Create Function FN_Get_MaxPayment

(f_Year integer = Null)

Returns decimal(6,2)

Language plpgsql

As \$body\$

Declare

v_Amount decimal(6,2);

Begin

If f_Year Is Null Then

Return 'Year was not passed in';

Else

Select Max(Amount)

Into v_Amount

From Payment

Where date_part('Year',paymentdate) = f_Year;

Return v_Amount;

End If;

End;

\$body\$;

select FN_Get_MaxPayment(2005);

Stored Procedure/Function Error Handling

Error Handling

Stored procedures in PostgreSQL can contain many different types of SQL statements including queries, Inserts, Updates, And Deletes. A DML (Data Manipulation Language) statement inside a stored procedure works exactly as it does outside of a stored procedure. However, there are some additional considerations when using DML statements inside stored procedures, especially when they are called from client applications (for example, Java, Python, or .NET). If an error occurs (such as a constraint violation or invalid data), PostgreSQL returns a detailed error message to the client. These messages are often technical and may not be meaningful to the end user. To make error handling more user-friendly, we can intercept any errors in our procedure and raise a clearer message.

Error Handling

Functions can include robust error handling even when they do not modify data. This is especially useful when performing calculations, data lookups, or conversions where invalid inputs or runtime issues may occur.

To handle errors gracefully in PostgreSQL, we use a `Begin ... Exception ... End` block. This structure lets us catch errors and raise more meaningful messages. When an error is handled in an `Exception` block, the PL/pgSQL local variables keep the values they held when the error happened, but any changes made to the database within that block are rolled back.

Error Handling

Simplified Syntax:

Begin

-- Code that may cause an error

Exception

When condition Then

-- Handle the error

End;

Error Handling

Examples of handling possible exceptions:

Begin

-- Code that may cause an error

Exception

When too_many_rows Then

Raise Exception 'Duplicate record found error: %', SQLERRM;

When no_data_found Then

Raise Exception 'Record not found error: %', SQLERRM;

When unique_violation Then

Raise Exception 'Key value not unique error: %', SQLERRM;

When others Then

Raise Exception 'Unknown error: %', SQLERRM;

End;

Notes

- SQLERRM contains the error message text (not usable outside exceptions)
- When “others” catches all remaining exceptions.
- Raise Notice – gives warning message to client (i.e. pgAdmin)
- Raise Exception – raises an error to client (i.e. pgAdmin) and stops processing

Some common integrity exception examples are `not_null_violation`, `foreign_key_violation`, `unique_violation`, `check_violation`.

Some common data exception examples are `division_by_zero`, `datetime_field_overflow`, `invalid_datetime_format`, `numeric_value_out_of_range`.

Example

Create a stored procedure that inserts a new club record. If an error occurs such as trying to insert a duplicate primary key, raise a clear error message.

```
Create Procedure PR_Add_Club
(p_ClubID varchar(10), p_ClubName varchar(50))
Language plpgsql
As $$
Begin
    Insert Into Club
        (ClubID, ClubName)
    Values
        (p_ClubID, p_ClubName);
    Raise Notice 'Club % added successfully', p_ClubID;
Exception
    When unique_violation Then
        Raise Exception 'A club with ID % already exists', p_ClubID;
    When others Then
        Raise Exception 'An unexpected error occurred inserting the club % Error: %', p_ClubID, SQLERRM;
End;
$$;
```

To run:

```
Call PR_Add_Club ('NewID', 'New Club Name');
```

```
Call PR_Add_Club ('COMP1017', 'New Club Name');
```

Example

Create a stored function that returns a student's ID when passed in a first name. If an error occurs, such as a duplicate or non-existent first name, raise clear error messages. You must use 'Select...Into Strict...' to ensure the exceptions fire if the first name is not unique or does not exist. Without 'Select...Into Strict...', the database would populate the local variable with the first occurrence if there were multiple first names, or null if there was a non-existent first name.

```
Create Function FN_Get_Staff_ID (p_FirstName varchar (50))
Returns integer
Language plpgsql
As $body$
Declare
    v_StaffID integer;
Begin
    Select StaffID
    Into Strict v_StaffID
    From Staff
    Where FirstName = p_FirstName;
    Raise Notice 'Unique staff first name % exists', p_FirstName;
    Return v_StaffID;
Exception
    When No_Data_Found Then
        Raise Exception 'Staff first name % not found', p_FirstName;
    When Too_Many_Rows Then
        Raise Exception 'Staff first name % is not unique',
p_FirstName;
    When Others Then
        Raise Exception 'Unknown error: %', SQLERRM;
End;
$body$
```

To run:

```
Select FN_Get_Staff_ID ('Cher');
```

Optional: recreate the Function
with a default value for first name:

```
Drop Function FN_Get_Staff_ID (p_FirstName varchar (50));
```

```
Create Function FN_Get_Staff_ID  
(p_FirstName      varchar (50) = null)  
Returns integer  
Language plpgsql  
As $body$  
Declare v_StaffID integer;  
Begin  
If p_FirstName Is Null Then  
Raise Exception 'You must enter a staff first name';  
Else  
Begin  
Select StaffID  
Into Strict v_StaffID  
From Staff  
Where FirstName = p_FirstName;  
Raise Notice 'Unique staff first name % exists', p_FirstName;  
Return v_StaffID;  
Exception  
When No_Data_Found Then  
Raise Exception 'Staff first name % not found', p_FirstName;  
When Too_Many_Rows Then  
Raise Exception 'Staff first name % is not unique', p_FirstName;  
When Others Then  
Raise Exception 'Unknown error: %', SQLERRM;  
End;  
End If;  
End;  
$body$
```

To run:

```
Select FN_Get_Staff_ID ('Cher');  
Select FN_Get_Staff_ID ('Bob');  
Select FN_Get_Staff_ID ();
```

Activity

Create a stored procedure to add a new course. Include exception handling for when the course already exists.

```
Create Procedure PR_Add_Course (p_CourseID Char(8), p_CourseName varchar(40), p_CourseHours smallint, p_MaxStudents integer, p_CourseCost decimal(6,2))
Language plpgsql
As $$
Begin
    Insert Into Course
        (CourseID, CourseName, CourseHours, MaxStudents, CourseCost)
    Values
        (p_CourseID, p_CourseName, p_CourseHours, p_MaxStudents, p_CourseCost);
    Raise Notice 'Course% added successfully', p_CourseID;
Exception
    When unique_violation Then
        Raise Exception 'A course with ID % already exists', p_CourseID;
    When others Then
        Raise Exception 'An unexpected error occurred inserting the course % Error: %', p_CourseID, SQLERRM;
End;
$$;
```

```
Call PR_Add_Course ('COMP1017', 'New Course', 80, 30, 500);
```

```
Call PR_Add_Course ('COMP1017'::Char(8), 'New Course'::varchar(40), 80::smallint, 30::integer, 500::decimal(6,2));
```


Activity

Create a function to return a StudentID if a last name exists in the student table. Display -404 if the last name does not exist, or -222 if the last name is not unique.

```
Create Function FN_Display_StudentID (p_LastName varchar(35) = null)
Returns decimal(8,0)
Language plpgsql
As $body$
Declare
    v_StudentID integer;
Begin
    if p_LastName is null then
        Raise Exception 'You must enter a student last name';
    else
        Begin
            Select StudentID
            Into Strict v_StudentID
            From Student
            Where LastName = p_LastName;
        -- Need 'Strict' to ensure the exceptions fire if the last name is not unique or does not exist
        -- Without 'Strict', the database would populate the variable with the first occurrence if there were multiple last names, or null if there was a non-existent last name
        Raise Notice 'Student last name % exists', p_LastName;
        Return v_StudentID;
    Exception
        When No_Data_Found Then
            Return -404;
        When Too_Many_Rows Then
            Return -222;
    End;
End If;
End;
$body$;

Select FN_Display_StudentID('Petroni');
Select FN_Display_StudentID('Kent');
Select FN_Display_StudentID('Henke');
```

Homework

Before next class please finish “Stored Function Exercises with Exceptions (PostgreSQL)” found in Brightspace in “Week 10 & 11 – Stored Procedures” under Activities.